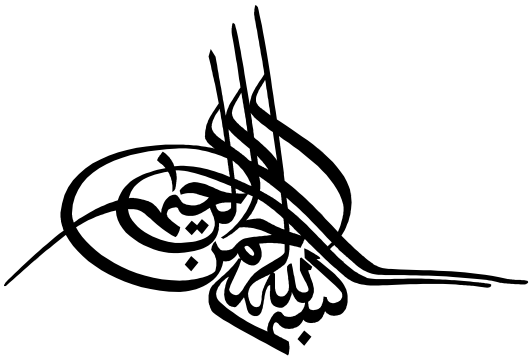


کامپایلر

سری کتاب‌های کمک آموزشی کارشناسی ارشد

مجموعه مهندسی کامپیوتر

مؤلف: لیلا امینی

سرشناسه	: امینی، لیلا، ۱۳۵۷
عنوان	: کامپایلر
مشخصات نشر	: تهران : مشاوران صعود ماهان، ۱۴۰۱
مشخصات ظاهری	: ۱۱۹ ص
فروست	: سری کتاب های کمک آموزشی کارشناسی ارشد
شابک	: ۹۷۸-۶۰۰-۳۸۹-۰۰۴-۶
وضعیت فهرست نویسی	: فیپای مختصر
یادداشت	:این مدرک در آدرس http://opac.nlai.ir قابل دسترسی است.
شماره کتابشناسی ملی	: ۳۷۷۲۲۰۳

نام کتاب: ..... کامپایلر

مولف:.....لیلا امینی

ناشر:.....مشاوران صعود ماهان

نوبت و تاریخ چاپ: .....اول / ۱۴۰۱

تیراژ: ..... ۱۰۰۰ نسخه

قیمت:..... ۱/۹۹۰/۰۰۰

شابک: ISBN۹۷۸-۶۰۰-۳۸۹-۰۰۴-۶

انتشـــارات مشاوران صعود ماهان: خیابان ولیعصر، بالاتر از تقـاطع مطهری،

روبروی قنادی هتل بزرگ تهـــران، جنب بانک ملی، پلاک ۲۰۵۰

تلفن: ۴- ۸۸۱۰۰۱۱۳

سخن ناشر

«ن والقلم و ما یسطرون»

کلمه نزد خدا بود و خدا آن را با قلم بر ما نازل کرد.

به پاس تشکر از چنین موهبت الهی، موسسه ماهان درصدد برآمده است تا در راستای انتقال دانش و مفاهیم با کمک اساتید مجرب و مجموعه کتب آموزشی خود برای شما داوطلبان ادامه تحصیل در مقطع کارشناسی ارشد گام موثری بردارد. امید است تلاش‌های خدمتگزاران شما در این موسسه پایه‌گذار گام‌های بلند فردای شما باشد.

مجموعه کتاب‌های کمک آموزشی ماهان به‌منظور استفاده داوطلبان کنکور کارشناسی ارشد سراسری و آزاد تالیف شده‌اند. در این کتاب‌ها سعی کرده‌ایم با بهره‌گیری از تجربه اساتید بزرگ و کتب معتبر داوطلبان را از مطالعه کتاب‌های متعدد در هر درس بی‌نیاز کنیم.

دیگر تالیفات ماهان برای سایر دانشجویان به‌صورت ذیل می‌باشد.

- **مجموعه کتاب‌های ۸ آزمون:** شامل ۵ مرحله کنکور کارشناسی ارشد ۵ سال اخیر به همراه ۳ مرحله آزمون تالیفی ماهان همراه با پاسخ تشریحی می‌باشد که برای آشنایی با نمونه سوالات کنکور طراحی شده است. این مجموعه کتاب‌ها با توجه به تحلیل ۳ ساله اخیر کنکور و بودجه‌بندی مباحث در هریک از دروس، اطلاعات مناسبی جهت برنامه‌ریزی درسی در اختیار دانشجو قرار می‌دهد.
- **مجموعه کتاب‌های کوچک:** شامل کلیه نکات کاربردی در گرایش‌های مختلف کنکور کارشناسی ارشد می‌باشد که برای دانشجویان جهت جمع‌بندی مباحث در ۲ ماهه آخر قبل از کنکور مفید می‌باشد.

بدین‌وسیله از مجموعه اساتید، مولفان و همکاران محترم خانواده بزرگ ماهان که در تولید و به‌روزرسانی تالیفات ماهان نقش موثری داشته‌اند، صمیمانه تقدیر و تشکر می‌نماییم.

دانشجویان عزیز و اساتید محترم می‌توانند هرگونه انتقاد و پیشنهاد درخصوص تالیفات ماهان را از طریق سایت ماهان به آدرس mahan.ac.ir با ما در میان بگذارند.

موسسه آموزش عالی آزاد ماهان

سخن مؤلف

کتابی که پیش رو دارید حاوی مجموعه کاملی از بهترین خلاصه منابع مختلف درس کامپایلر براساس سرفصل‌های مصوب شورای عالی برنامه‌ریزی، تست‌های تالیفی، نکات ویژه و سوالات کنکور کارشناسی ارشد سال‌های گذشته، همراه با پاسخ تشریحی ارائه شده است.

پیش‌بینی می‌شود با مطالعه این مجموعه ضمن صرفه‌جویی در وقت با مفاهیم و نکات ضروری آشنا شده که این مهم با توجه به استقبال خوب و تماس‌های تشکرآمیز خوانندگان محترم در این مدت بر ما اثبات شده است.

یقینا هر اثری توام با خطاهایی است به‌همین‌منظور برای رفع آنها در انتظار دریافت نظرات شما استادان بزرگوار، دانشجویان و دیگر خوانندگان هستیم.

با تشکر

لیلا امینی

فهرست

عنوان | صفحه

فصل اول

آشنایی با کامپایلر و تحلیلگر لغوی

- وظایف کامپایلر
- طبقه‌بندی کامپایلرها
- مراحل کامپایل
- مدیریت جدول نمادها
- کامپایلر و مفسر
- گرامرها
- مراحل ساخت درخت ترجمه
- عبارات منظم
- مروری بر نظریه زبان‌ها و ماشین‌ها
- دیاگرام‌های انتقال
- تبدیل NFA به DFA

آشنایی با کامپایلر و تحلیلگر لغوی

یک برنامه نوشته شده در یک زبان سطح بالا برای قابل اجرا بودن باید به برنامه‌ای قابل فهم توسط ماشین تبدیل شود.

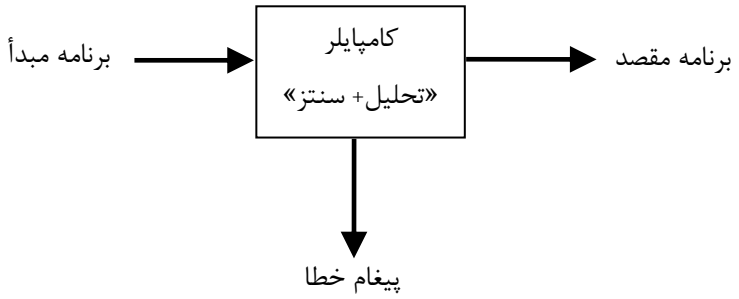
کامپایلر (Compiler) برنامه‌ای است که یک برنامه نوشته شده در یک زبان به‌نام زبان منبع (Source Language) را به برنامه‌ای معادل به زبان دیگر به‌نام زبان مقصد (Target Language) تبدیل می‌کند.

بدین ترتیب برای اجرای یک برنامه نوشته شده به زبان سطح بالا ابتدا برنامه کامپایل شده و در نتیجه آن برنامه مقصد به‌دست می‌آید، سپس برنامه مقصد در حافظه بار شده و اجرا می‌گردد. علاوه‌براین کامپایلرها در حین کامپایل، درصورت وجود خطا در برنامه مبدا، آن را به کاربر نمایش می‌دهند.

وظایف کامپایلر

۱- ترجمه برنامه از زبان مبدأ به برنامه معادل در زبان‌های میانی همانند اسمبلی

۲- گزارش وجود خطاها در برنامه مبدأ به کاربر



طبقه‌بندی کامپایلرها

دسته‌بندی کامپایلرها براساس چگونگی ساخت و عملیات:

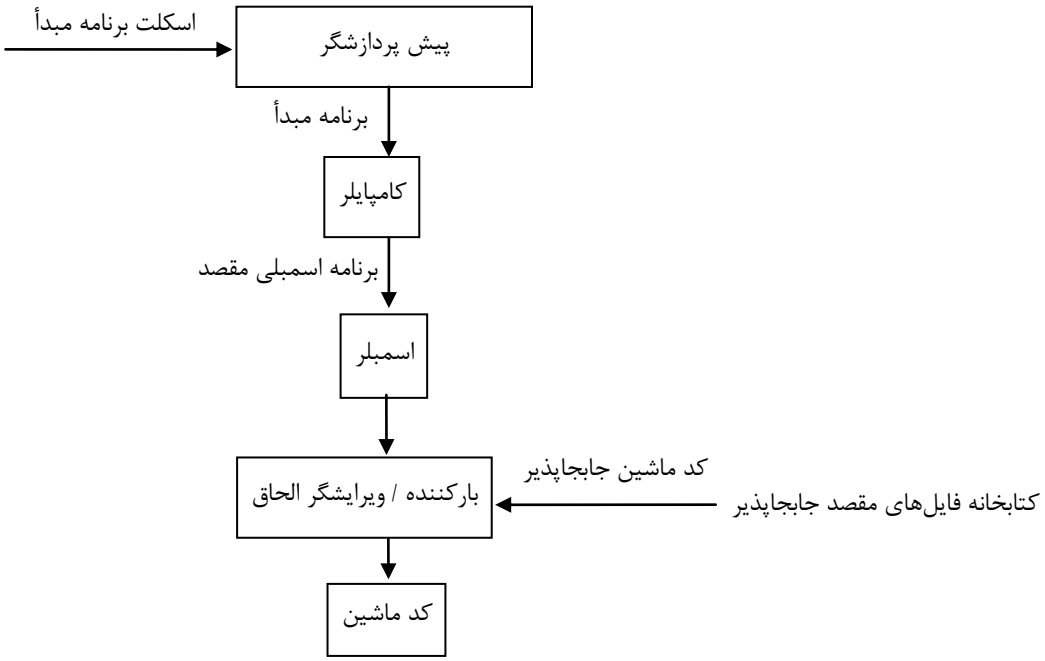
- تک‌گذره
- چندگذره
- اشکال‌زدا و Load-and-go
- بهینه‌ساز

عملیات کامپایلر

عملیات کامپایلر از دو بخش تشکیل شده است که عبارتند از:
۱- بخش تحلیل که وظایف آن عبارت است از:
تجزیه برنامه مبدأ به اجزای تشکیل‌دهنده‌اش و تولید کد میانی از برنامه مبدأ
۲- بخش سنتز (ترکیب) که وظیفه آن عبارت است از ساخت برنامه مقصد توسط کد میانی

سیستم پردازش زبان

سیستم پردازش زبان از بخش‌های زیر تشکیل شده است:
پیش پردازشگر، کامپایلر، اسمبلر، بارکننده و ویرایشگر الحاق
وظایف بخش پیش پردازشگر عبارت است از:
۱- جمع‌آوری ماژول‌های برنامه مبدأ موجود در فایل‌های جداگانه
۲- تبدیل بخش‌های خلاصه شده به نام درشت دستورات به احکام زبان مبدأ
ارتباطات در سیستم پردازش زبان به‌صورت زیر می‌باشد:



فازهای تحلیل در عمل کامپایل

سه فاز تحلیل در عمل کامپایل عبارتند از:

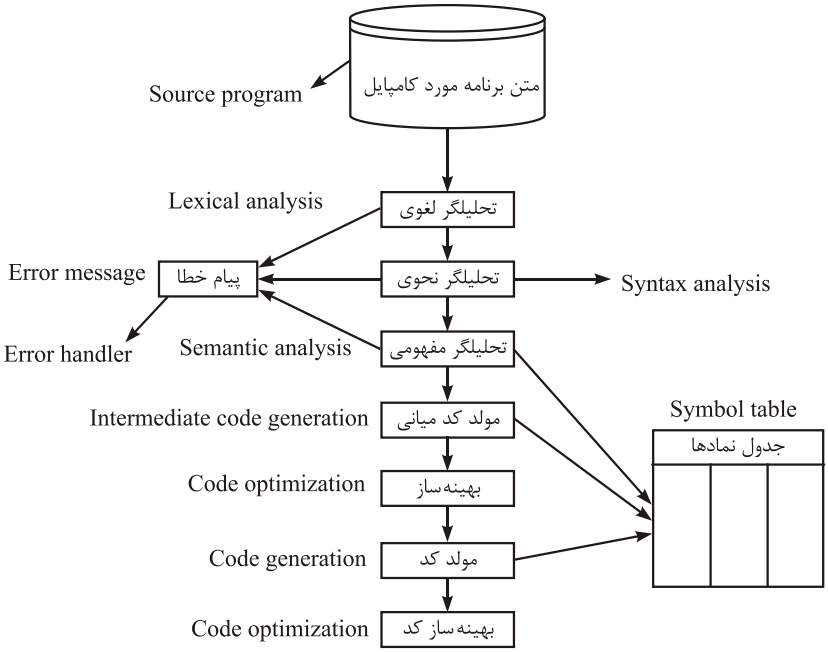
۱- تحلیل خطی (تحلیل لغوی یا پویش)	تشخیص نشانه‌ها
۲- تحلیل سلسله مراتبی (تحلیل نحوی یا تجزیه)	گروه‌بندی نشانه‌های برنامه مبدأ به جملات گرامری
۳- تحلیل معنایی	بررسی خطاهای معنایی برنامه

مراحل کامپایل

عمل کامپایل در شش مرحله زیر صورت می‌گیرد:

۱- تحلیل لغوی (Lexical analysis)
۲- تحلیل نحوی (Syntax analysis)
۳- تحلیل معنایی (Semantic analysis)
۴- تولید کد میانی (Intermediate code generation)
۵- بهینه‌سازی کد (Code optimization)
۶- تولید کد نهایی (Code generation)

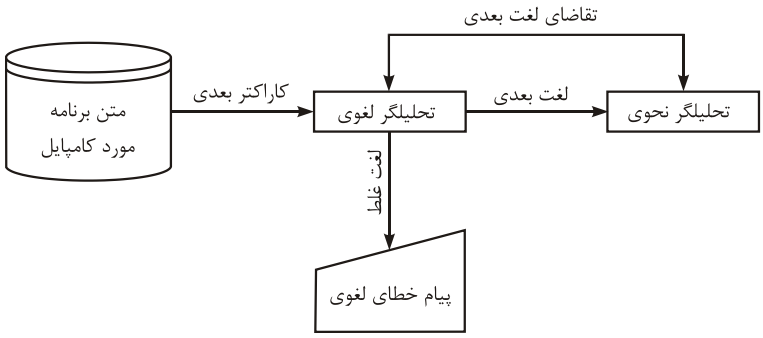
این شش مرحله به همراه بخش‌های «پردازش خطا» و «جدول سمبل‌ها» یک کامپایلر را تشکیل می‌دهند. ارتباط بین این مراحل در شکل زیر نمایش داده شده است. به سه بخش اول و قسمتی از بخش چهارم که به زبان مبدأ وابسته است، جلوبندی (Forward end) و به بقیه بخش‌ها که به زبان مقصد وابسته است، عقب بندی (Back end) گفته می‌شود.
در بعضی از مراجع بعد از تولید کد نهایی، یک بهینه‌ساز دیگر برای کد استفاده می‌شود.



تحلیلگر لغوی

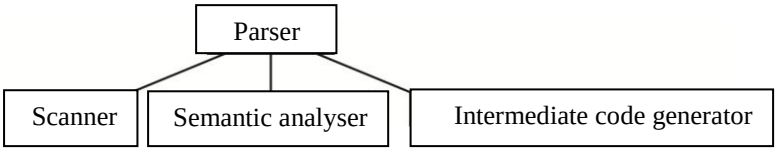
وظیفه تابع تحلیلگر لغوی (Lexical Analyser) در متن برنامه مورد کامپایل است، بدین ترتیب که برنامه ورودی را نویسه به نویسه (Character to character) می‌خواند و به‌دنباله‌ای از نشانه‌ها (Tokens) تبدیل می‌کند. با هر بار فراخوانی لغت بعدی را از متن برنامه تشخیص می‌دهد و اطلاعات لازم در مورد لغت را برای تحلیلگر نحوی ارسال می‌دارد. توکن‌های خروجی، تحلیلگر لغوی می‌باشند و در جدول سمبل‌ها با فرمت خاصی ذخیره می‌شوند. همچنین به‌عنوان ورودی تحلیلگر نحوی استفاده می‌شوند. به‌عبارت دیگر تحلیلگر لغوی واسط بین برنامه مبدأ و تحلیلگر نحوی است.
انواع مختلف توکن‌ها عبارتند از:
کلمات کلیدی (Keywords)، عملگرها (Operators)، ثابت‌ها (Literals)، شناسه‌ها (Identifiers) و جداکننده‌ها (Delimiters)، بنابراین به اسامی متغیرها، توابع، رویه‌ها و به‌طورکلی اسامی که کاربر انتخاب می‌کند، گفته می‌شود.
جداکننده‌ها شامل Space، Tab و Enter می‌باشد.
در اغلب زبان‌های برنامه‌سازی این کلمات کلیدی رزرو شده‌اند و کاربر مجاز نیست از هیچ‌کدام از آن‌ها به‌عنوان اسم متغیر تابع استفاده کند.

به واحدی از کامپایلر که کار تحلیل واژه‌ای را انجام می‌دهد اسکنر Scanner گویند.
تحلیلگر لغوی توضیحات (Comment) را حذف می‌کند و فاصله‌هایی که کاراکترهای شناسه‌ها را از هم جدا می‌کند، حذف می‌کند و اگر خطایی رخ دهد آن را گزارش می‌دهد.



تحلیلگر نحوی

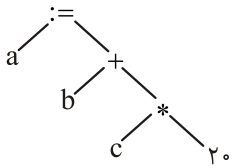
تحلیل سلسله مراتبی، تجزیه یا تحلیل نحوی (Syntax analysis) نامیده می‌شود. وظیفه تحلیلگر نحوی تشخیص صحت فرم ظاهری برنامه‌ها از لحاظ دستورالعمل زبان برنامه‌سازی مربوطه است. تحلیلگر نحوی با داشتن گرامر مربوط به زبان و Tokenهای دریافتی از تحلیلگر لغوی، برای هر دستور درخت اشتقاق آن را می‌سازد و آن را به‌عنوان خروجی به تحلیلگر مفهومی می‌دهد. اگر دستور متعلق به زبان مربوطه نباشد خطا گزارش می‌دهد. به واحدی از کامپایلر که کار تحلیل نحوی را انجام می‌دهد پارسر (Parser) می‌گویند. پارسر اصلی‌ترین بخش کامپایلر است و به‌عنوان مدیر کامپایلر، اسکنر، آنالیز معنایی و تولید کد میانی را به عهده دارد.



در مرحله تحلیل نحوی، درخت تجزیه (Parse tree) تولید می‌شود. ساده شده آن فاقد ناپایانه‌هاست، درخت نحو (Syntax tree) نام دارد.

مثال:

$a := b + c * 20$



مثال: اگر در زبان C دستوری به شکل $for(i = 0; i + +)$ داشته باشیم خطای نحوی رخ می‌دهد.

تحلیلگر معنایی (مفهومی)

مهم‌ترین وظیفه تحلیلگر معنایی کنترل نوع (Type checking) است. اگر تبدیل نوع مجاز باشد عمل تبدیل نوع را انجام می‌دهد وگرنه خطای تبدیل نوع را گزارش می‌دهد. تحلیلگر مفهومی براساس نوع اسامی و متغیرها که در جدول نمادها مشخص شده، صحت استفاده آن‌ها را مورد آزمون قرار می‌دهد. اخطارها (Warning) به این مرحله تعلق دارند.

$float + int \rightarrow$ تبدیل نوع انجام می‌شود.

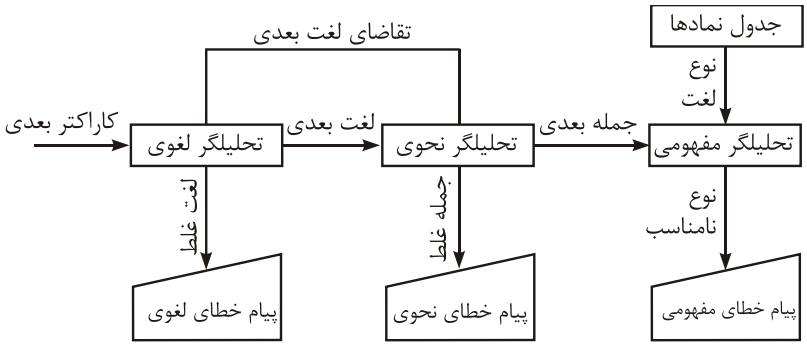
$float + char \rightarrow$ خطا

در مورد تقسیم بر صفر، اگر صفر صراحتاً در عبارت آمده باشد، در پارسر قابل تشخیص است. اما اگر عبارتی به‌صورت $5/(3-2-1)$ داشته باشیم در تحلیلگر معنایی معلوم می‌شود.

دسترسی به اندیس غیرمجاز در آرایه، عدم تطابق نوع و تعداد پارامترهای توابع در فراخوانی و تعریف، در تحلیلگر معنایی شناسایی می‌شود.

عبارت $\frac{6}{i}$ در کامپایل تشخیص داده نمی‌شود.

$$i = 0 \Rightarrow \text{Run time error}$$



تولیدکننده کد میانی

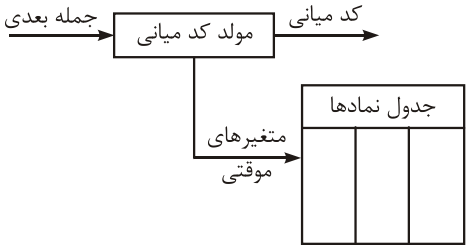
در این بخش برنامه ورودی به یک زبان میانی تبدیل می‌شود. این زبان میانی باید خصوصیات زیر را داشته باشد:

۱- تولید آن آسان باشد.

۲- به آسانی به زبان مقصد ترجمه شود.

این کد میانی شرایط خوبی را برای بهینه‌سازی کد برنامه‌ها فراهم می‌کند و به‌راحتی قابل حمل روی سخت‌افزار با کد و ساختار متفاوت است.

کد میانی خیلی به زبان ماشین نزدیک است اما از هرگونه ساختار و جزئیات ماشینی خاص، مستقل است.



بهینه‌سازی کد میانی

در این بخش سعی می‌شود تا کد میانی تقلیل حجم یافته و به‌صورتی درآید که سریع‌تر انجام شود. دلیل اینکه عمل بهینه‌سازی همراه با کد میانی انجام نمی‌شود این است که عمل تولید کد میانی همراه با درخت انجام می‌شود و کارکردن با درخت مشکل است ولی بهینه‌ساز کد، یک فایل را به‌عنوان ورودی می‌گیرد.

تولیدکننده کد نهایی

کد نهایی می‌تواند به فرم کد ماشین (Machine Code) یا کد افزونه (Augmented Code) یا کد مجازی (Virtual Code) باشد.

کد ماشین: بدون نیاز به هیچ کار دیگری، قابل اجراست.

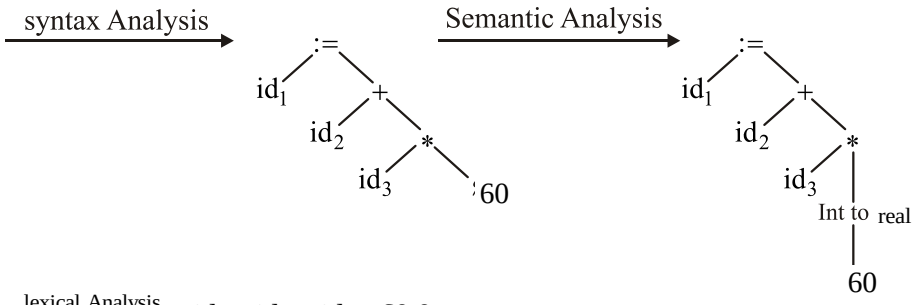
کد افزونه: یک سری کد برای انجام کارهای کنترلی نظیر درخواست حافظه و... از سیستم عامل به کد ماشین اضافه می‌کند.

کد مجازی: در سطح بالاتری است و وابسته به سیستم عامل و معماری نیست.

کد نهایی از لحاظ آدرس‌دهی حافظه، تقسیم‌بندی دیگری نیز دارد:

- کد اسمبلی (Assembly Code): در آدرس‌های سطح بالا وجود دارد.
- تصویر حافظه (Memory Image): دقیقاً آدرس‌های حافظه را دارد.
- کد جابجاپذیر (Relocatable Code): آدرس‌های نسبی را دارد.

شکل زیر مراحل کامپایل $p=i+r*60$ و تأثیر هر مرحله از کامپایل را روی این دستور نشان می‌دهد. (p,i,r ، real هستند)



$$p:=i+r*60 \xrightarrow{\text{lexical Analysis}} id_1=id_2+id_3*60.0$$

$$\xrightarrow{\text{intermediate code Generation}} \begin{cases} t_1:=\text{int to real}(60) \\ t_2:=id_3*t_1 \\ t_3:=id_2+t_2 \\ id_1:=t_3 \end{cases} \xrightarrow[\text{optimizer}]{\text{Code}} \begin{cases} t_1=id_3*60.0 \\ id_1=id_2+t_1 \end{cases} \xrightarrow{\text{Code Generator}} \begin{cases} MOV\ R_1,id_3 \\ MUL\ R_1,60.0 \\ MOV\ R_2,id_2 \\ ADD\ R_1,R_2 \\ MOV\ id_1,R_1 \end{cases}$$

مدیریت جدول نمادها (Symbol table manager)

ساختمان داده‌ای است که شامل رکوردی برای هر نام متغیر، به همراه فیلدهایی برای صفات آن است.

این صفات، ممکن است اطلاعاتی درباره حافظه تخصیص‌یافته برای نام، نوع آن، حوزه آن (جایی در برنامه که مقدارش می‌تواند استفاده شود) و ... باشد که باید متغیرهای استفاده شده در برنامه و اطلاعات آن‌ها در جایی ثبت شود. یک جدول نماد، ساختمان داده‌ای است که برای هر متغیر، یک رکورد در نظر می‌گیرد. هر شناسه‌ای که تحلیلگر لغوی تشخیص می‌دهد به این جدول وارد می‌شود و البته در مراحل بعدی صفات مربوط به این شناسه وارد جدول می‌شود. وظیفه این بخش امکان یافتن سریع رکورد شناسه به منظور ذخیره و بازیابی داده‌هایش است.

خطاپرداز (Error handler)

فاز تحلیل لغوی قادر به شناسایی خطاهایی است که در نتیجه آن‌ها، کاراکترهای باقی مانده ورودی، نشانه‌ای از زبان را تشکیل نمی‌دهند. هر بار که خطایی در یکی از مراحل پیش بیاید، رویه‌ای به نام خطاپرداز فراخوانده می‌شود. معمولاً پارسر و اسکنر، بیشتر خطاهایی را که در یک برنامه ممکن است وجود داشته باشد، تشخیص می‌دهند.

ابتدا (Front – end) و انتهای (Back – end) کامپایلرها:

به چهار مرحله اول کامپایلرها و بخشی از مرحله بهینه‌سازی که بستگی به زبان منبع (Source Program) دارد و مستقل از زبان ماشین است، Front – end کامپایلر می‌گویند.

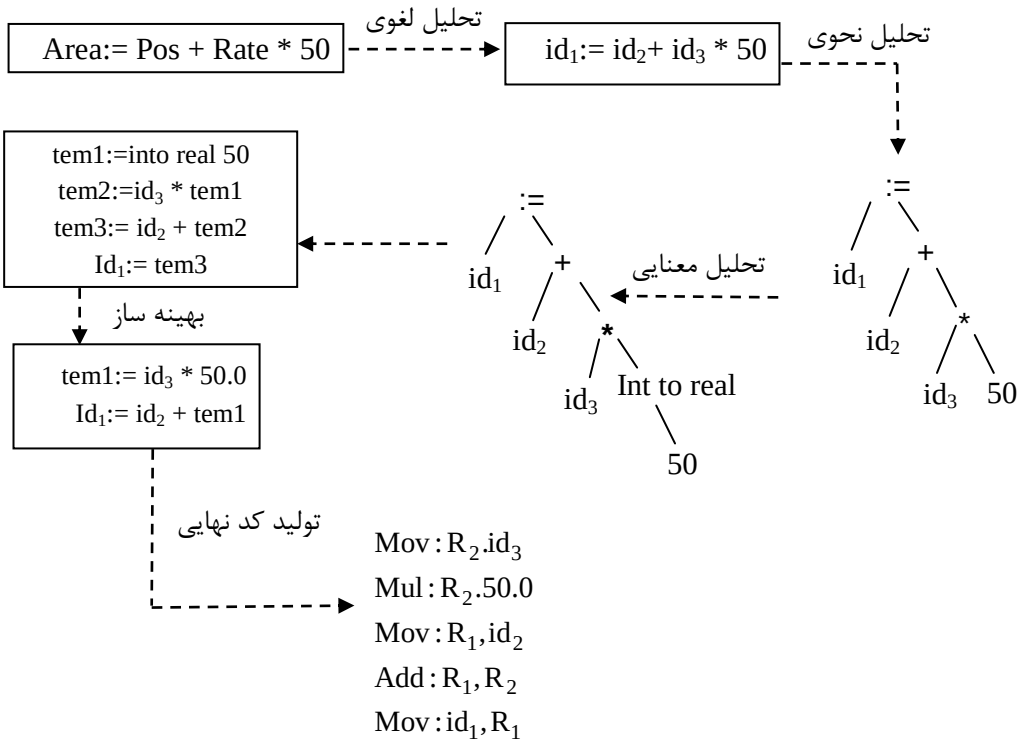
به بخشی از مرحله بهینه‌سازی و مرحله آخر کامپایل که وابسته به زبان ماشین است، Back end کامپایلر گویند. معمولاً این

بخش از کامپایلر وابسته به زبان منبع نیست. بخش Front – end یک کامپایلر برای تمام ماشین‌ها یکسان است اما Back – end برای ماشین‌های مختلف، متفاوت است.

کامپایلر و مفسر

کامپایلر یک‌بار ترجمه می‌کند و بارها اجرا می‌شود اما مفسر برای هر بار اجرا، یک‌بار تفسیر می‌کند. به‌عنوان مثال یکی از نقاط ضعف مفسرها نسبت به کامپایلرها حلقه‌ها هستند. کامپایلرها یک حلقه 10 تایی را 1 بار ترجمه و 10 بار اجرا می‌کند ولی مفسر همین حلقه را 10 بار ترجمه و 10 بار هم اجرا می‌کند، بنابراین overhead بالایی دارد. بعد از کامپایل مطمئن هستیم که خطای نحوی نداریم، اما چون همه خطوط تفسیر نمی‌شوند در مورد مفسر این اطمینان وجود ندارد. ایجاد تغییر در برنامه با زبان مفسری بسیار ساده‌تر است و تصحیح خطا ساده‌تر انجام می‌شود. مفسر معمولاً به زبان سطح بالا نوشته می‌شود و روی اغلب ماشین‌ها قابل اجراست در حالی‌که کد مقصدی که توسط کامپایلر تولید می‌شود، در ماشین خاص اجرا می‌شود. به‌عبارت دیگر استفاده از مفسر قابلیت حمل برنامه را افزایش می‌دهد. مفسرها کد میانی تولید نمی‌کنند یعنی شامل بخش Intermediate Code Generator نیستند.

مثال: مراحل کامپایل عبارت Area:= Pos + Rate * 50 به‌صورت زیر می‌باشد:



گرامرها (Grammars)

گرامر: مجموعه متناهی از قواعد که ساختار جملات یک زبان را بیان می‌کند از آنجایی‌که گرامر در ارتباط با عمل ترجمه اهمیت زیادی دارد. به معرفی اجمالی گرامرها در این بخش می‌پردازیم.

گرامرها توسط چهارتایی $G = (N, T, S, P)$ نشان داده می‌شوند که در آن:

N: مجموعه ناپایانه‌ها (Non Terminals) که با حروف بزرگ نشان می‌دهند.

T: مجموعه پایانه‌ها (Terminals) که با حروف کوچک نشان می‌دهند.

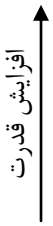
S: علامت شروع گرامر (Starting Symbol)

P: مجموعه‌ای متناهی از قواعد تولید (Production Rules)

عملگری که برای تولید رشته‌های زبان از طریق یک گرامر به‌کار می‌رود عملگر اشتقاق است. برای نمایش مراحل اشتقاق در برخی

از گرامرها می‌توان از درخت تجزیه یا درخت (Derivation tree) استفاده نمود. ریشه درخت تجزیه معمولاً یکی از ناپایانه‌های گرامر (اغلب نماد شروع) و گره‌های آن می‌توانند هر یک از عناصر باشند. با کنار هم قرار دادن برگ‌های این درخت از چپ به راست محصول آن به‌دست می‌آید که همان جمله‌ای است که از ناپایانه واقع در ریشه درخت مشتق شده است.

انواع گرامرها



گرامرها طبق تقسیم‌بندی چامسکی در چهار گروه عمده طبقه‌بندی می‌شوند:

۱- گرامرهای نوع صفر (گرامرهای بدون محدودیت Unrestricted)

۲- گرامرهای نوع یک (گرامرهای حساس به متن Context Sensitive)

۳- گرامرهای نوع دو (گرامرهای مستقل از متن Context free)

۴- گرامرهای نوع سه (گرامرهای منظم Regular)

گرامرهای نوع صفر محدودیت خاصی ندارند و قواعدی به فرم $\alpha \to \beta$ هستند که سمت چپ این قواعد باید شامل حداقل یک ناپایانه باشد.

$\alpha \neq \Lambda(\text{null})$

در گرامرهای حساس به متن علاوه بر محدودیت فوق، طول سمت چپ یک قاعده باید کوچک‌تر یا مساوی با طول سمت راست باشد.

$|\alpha| \leq |\beta|$

در گرامرهای مستقل از متن علاوه بر محدودیت‌های فوق باید سمت چپ قواعد این گرامرها فقط یک ناپایانه باشد.

$\alpha \in N, |\alpha| = 1$

در گرامرهای منظم علاوه بر محدودیت‌های بالا، داریم:

$|\beta| = 1, \beta \in T$ یا $|\beta| = 2, \beta = aA$ منظم از راست

$|\beta| = 1, \beta \in T$ یا $|\beta| = 2, \beta = Aa$ منظم از چپ

فرم BNF (Backus – Nour Form)

برای نوشتن گرامرهای مستقل از متن به‌کار می‌رود. در این فرم ناپایانه‌ها بین دو علامت $< - >$ قرار می‌گیرند. به‌جای علامت $\to$ از علامت $::=$ استفاده می‌شود.

$< E > ::= < E > + < E >$

اسکنر را توسط زبان منظم و پارسر را توسط زبان مستقل از متن پیاده‌سازی می‌کنند.

اشتقاق (Derivation)

منظور از اشتقاق به‌دست آوردن یک جمله توسط قواعد تولید یک گرامر با شروع از ناپایانه می‌باشد که به دو نوع زیر تقسیم می‌شود:

- اشتقاق از چپ(LMD): همواره چپ‌ترین غیرپایانی جایگزین می‌شود.
- اشتقاق از راست(RMD): همواره راست‌ترین غیرپایانی در مراحل اشتقاق جایگزین می‌شود.

مثال: گرامر زیر را در نظر بگیرید:

$E \to E + E$

$E \to E * E$

$E \to (E)$

$E \to id$

با توجه به این گرامر، برای تولید جمله $id + id * id$ می‌توان از اشتقاق چپ و اشتقاق راست زیر استفاده نمود:

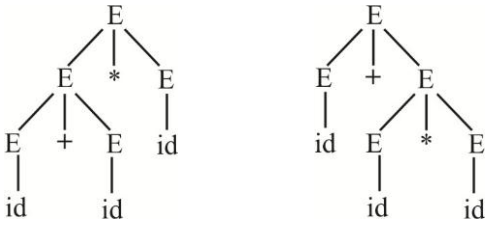
$$E \underset{L}{\Rightarrow} E + E \underset{L}{\Rightarrow} id + E \underset{L}{\Rightarrow} id + E * E \underset{L}{\Rightarrow} id + id * E \underset{L}{\Rightarrow} id + id * id$$

$$E \underset{R}{\Rightarrow} E + E \underset{R}{\Rightarrow} E + E * E \underset{R}{\Rightarrow} E + E * id \underset{R}{\Rightarrow} E + id * id \underset{R}{\Rightarrow} id + id * id$$

اگر اشتقاق به یکی از دو روش نامبرده انجام شود، اشتقاق قانونمند (Cononical) انجام شده است. معادل هر اشتقاق یک درخت تجزیه یا درخت اشتقاق داریم. درخت تجزیه نشان‌دهنده چگونگی اشتقاق رشته‌ای از زبان از نماد شروع گرامر می‌باشد.

مراحل ساخت درخت تجزیه به‌صورت زیر می‌باشد:

- S ریشه درخت می‌باشد.
- قانون $A \to XYZ$ به‌این‌صورت تبدیل می‌شود که A گره‌ای در درخت و Z,Y,X فرزندان آن می‌شوند.
- قانون $A \to Xa$ به‌این‌صورت تبدیل می‌شود که A گره‌ای در درخت و a,X فرزندان آن می‌شوند.
- پایانه‌ها (حروف کوچک) تنها در برگ‌ها دیده می‌شوند.



تفاوت دو درخت در اولویت عملگرهاست.

گرامر مبهم (Ambiguous Grammar)

به یک گرامر مستقل از متن، مبهم گویند اگر با استفاده از قواعد گرامر بتوانیم حداقل یک رشته پیدا کنیم که برای آن حداقل دو LMD یا حداقل دو RMD یا حداقل دو درخت پارس داشته باشیم. گرامری که در مثال قبل بررسی شد مبهم می‌باشد.

مثال: رفع ابهام گرامر مثال قبل به‌صورت زیر می‌باشد:

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $T \to (E)$
(6) $F \to id$

گرامر کاهش‌یافته (Reduced Grammar)

به یک گرامر مستقل از متن کاهش‌یافته گویند، اگر سه شرط زیر را دارا باشد:

۱- فاقد دستوراتی مانند $A \to A$ باشد.

۲- تمام غیرپایانی‌ها فعال (Active) باشند؛ یعنی حداقل در ایجاد یک جمله از زبان شرکت داشته باشند.

۳- تمام ناپایانه‌ها قابل دسترس (Reachable) باشند.

مثال: گرامر کاهش‌یافته زیر را در نظر بگیرید:

$A \to Ba$
$B \to b \mid Cb$
$C \to aCb \mid Dd$
$D \to \varepsilon$

$Active = \{A, B, D\}$
$A \to Ba$
$B \to b$
$D \to \varepsilon$

$Reachable = \{A, B\}$
$A \to Ba$
$B \to b$

نشان‌گذاری (نمایش پسوندی)

نشان‌گذاری یک عبارت مانند E به‌صورت زیر انجام می‌شود:

۱- اگر E متغیر و یا ثابت باشد، نشان‌گذاری آن برابر با خود E می‌شود.

۲- اگر E عبارتی به شکل E1 Op E2 باشد که Op عملگر دودویی است، نشان‌گذاری آن عبارت است از F1 F2 Op که F1, F2 نشان‌گذاری پسوندی E1 , E2 هستند.

۳- اگر E عبارتی دارای پرانتز به شکل (E1) باشد، آنگاه نشان‌گذاری برای E1 همان نشان‌گذاری برای E می‌باشد.

$\boxed{(9-5)+2} \Rightarrow \boxed{95-2+}$

ویژگی‌های تعریف نحوگرا

- کاربرد برای ترجمه ساختارهای زبان
- ترجمه ساختار بر حسب صفات مربوط به مؤلفه‌های نحوی
- تعیین نوع ساختار، مکان اولین دستور تولیدشده در برنامه هدف یا تعداد دستورات برای کامپایلر

تعریف نحوی جهت‌دار

شامل گرامر و مجموعه‌ای از قواعد معنایی وابسته به آن می‌باشد. صفات و محاسبه آن‌ها به‌صورت زیر انجام می‌شود:

- هر نماد در گرامر مجموعه‌ای از صفات را دارد.
- در هر مولد یا قانون گرامر مجموعه‌ای از قواعد معنایی برای محاسبه مقادیر صفات نمادها وجود دارد.

مراحل ساخت درخت ترجمه

مراحل ساخت درخت ترجمه به‌صورت زیر می‌باشد:

۱- ساختن درخت تجزیه برای ورودی

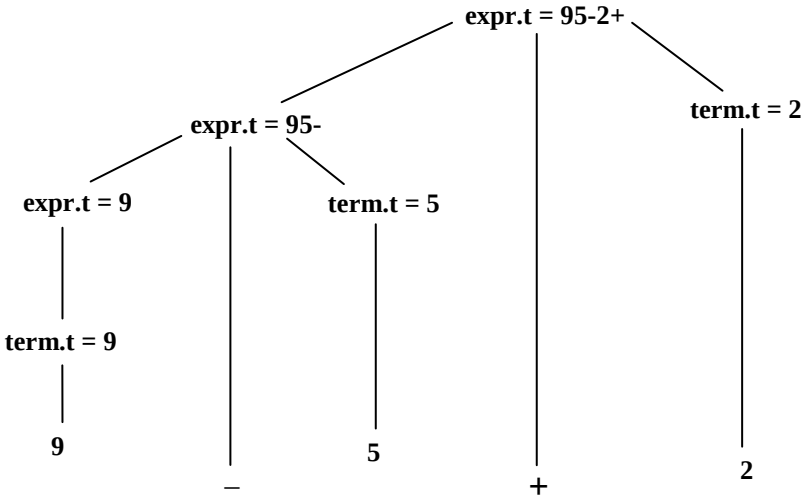
۲- اگر گره n در درخت تجزیه با نماد x از گرامر متناظر باشد، x.a مقدار صفت a از نماد x در آن گره است.

۳- مقدار x.a در گره n با استفاده از قواعد معنایی برای صفت a همراه با قانون گرامری استفاده شده در گره n محاسبه می‌شود.

مثال: با استفاده از قواعد فوق، تعریف نحوگرا برای ترجمه عبارات میانوندی به‌عبارت معادل پسوندی را به‌صورت زیر به‌دست می‌آوریم:

قانون	قواعد معنایی
expr → expr1 + term	expr.t = expr1.t ‖ term.t ‖ ‘+’
expr → expr1 – term	expr.t = expr1.t ‖ term.t ‖’ – ‘
expr → term	expr.t= term.t
term → 0	term.t = ‘0’
term → 1	term.t = ‘1’
…..	…..
term → 9	term.t = ‘9’

مثال: نمونه‌ای از یک درخت نحوی در زیر آمده است:



انواع درخت نحوی عبارتند از:

۱- درخت نحو مجرد: در این نوع درخت نحوی هر گره نماینده یک عملگر و فرزندان آن عملوند آن می‌باشد.

۲- درخت نحو واقعی: در این نوع درخت نحوی عملگرها خود فرزند محسوب می‌شوند.

الگوی ترجمه

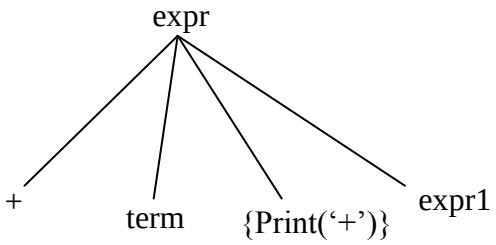
الگوی ترجمه، گرامر مستقل از متنی است که قطعه برنامه‌هایی که عملیات معنایی نامیده می‌شوند، در سمت راست قوانین آن اضافه شده‌اند و تفاوت آن با ترجمه نحوگرا، در نمایش ترتیب ارزشیابی قوانین به‌طور صریح می‌باشد.

طریقه ساخت درخت تولید شده برای الگوی ترجمه

برای این کار از رویه زیر استفاده می‌کنیم:

۱- ساختن فرزند اضافی برای درخت

۲- متصل نمودن این فرزند به گره مربوط به قانون خود در گرامر



یک برگ اضافی ساخته شده برای عمل معنی

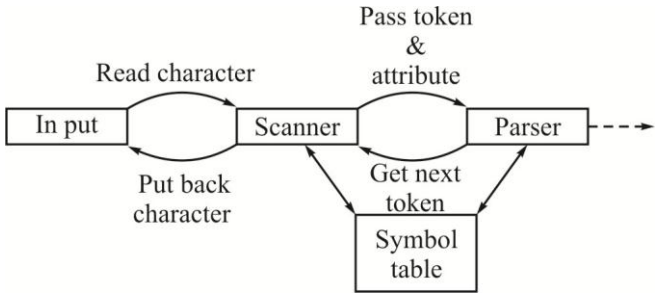
تحلیلگر لغوی

اولین فاز کامپایلر می‌باشد و وظیفه آن tokenize کردن است. هر بار که تحلیلگر نحوی احتیاج به‌داشتن نشانه بعدی داشته باشد، تحلیلگر لغوی را صدا می‌زند.

وظایف دیگر اسکنر عبارتند از:

- حذف فضاهای خالی و توضیحات (Comment)
- گزارش خطاها

از آنجا که اسکنر برنامه را به‌صورت کاراکتر به‌کاراکتر می‌خواند، می‌تواند تعداد کاراکترهای هر خط و به تبع آن شماره خطوطی که توکن در آن‌ها قرار دارد را نیز مشخص کند. تحلیلگر لغوی می‌تواند همراه با هر پیام خطا یک شماره سطر نیز چاپ کند. افزودن کدهایی که توسط include به برنامه اضافه می‌شوند، به‌عهده اسکنر است.



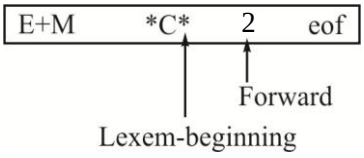
اسکنر علاوه بر اینکه تشخیص می‌دهد که توکن یک شناسه است، آدرس آن در جدول نشانه‌ها را نیز برای پارسر می‌فرستد.

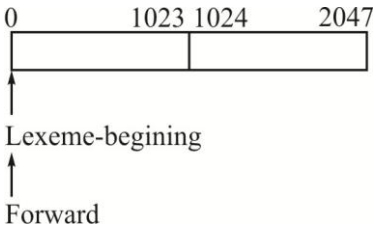
به آخرین توکنی که اسکنر یافته است، علامت پیش‌بینی (Lockahead symbol) و یا توکن جاری گفته می‌شود.

به فرم کلی که یک توکن می‌تواند داشته باشد، الگوی آن توکن می‌گوییم.
به دنباله‌ای از نویسه‌ها که تشکیل یک توکن می‌دهند، واژه (Lexem) آن توکن می‌گویند.
به‌عبارت دیگر برای هر نوع token یک Pattern وجود دارد، مثلاً Pattern علائم مقایسه: <, >, <=, => , ==
جدول زیر حاوی چند نمونه الگو و واژه است.

token	lexeme	Pattern
Const	Const	Const
If	if	If
relation	<,>,<=,=,>=,<>	< or > or <= or >= or <> or =
Id	pi	با حرف الفبا شروع و به‌دنبال آن حرف و رقم قرار می‌گیرد.
Num	3.141	هر ثابت عددی
Literal	"Core dumped"	هر رشته یا نویسه‌ای که بین دو علامت " قرار گیرد.

گاهی اوقات اسکنر برای اینکه تشخیص درستی دهد و توکن درست را به پارسر بفرستد، نیاز دارد چند کاراکتر دیگر از ورودی را بخواند. به‌عنوان مثال اگر اسکنر علامت ">" را ببیند، نیاز دارد که کاراکتر ورودی بعدی را نیز بخواند که اگر این کاراکتر "=" بود، توکن '>=' و در غیر این‌صورت توکن ">" را تشخیص داده و به پارسر بفرستد و کاراکتر اضافی را نیز به ورودی Put back می‌کند.
دلایلی که بهتر است پارسر و اسکنر به‌طور مجزا پیاده‌سازی شوند عبارتند از: سادگی، کارایی بالاتر و قابلیت حمل.
سادگی: پارسری که دربردارنده قواعد مربوط به حذف فضاهای خالی و توضیحات می‌باشد پیچیده‌تر از پارسری است که در کامپایلری قرار دارد که این اعمال را اسکنر آن انجام می‌دهد. زمان برترین کار کامپایلر کاری است که در اسکنر انجام می‌شود، پس باید سرعت کار آن را افزایش دهیم که این کار معمولاً با استفاده از بافر انجام می‌شود یعنی اسکنر برای تشخیص نهایی توکن‌ها و مطابقت دادن آن‌ها با الگوهای موجود چندین کاراکتر جلوتر را نیز مورد بررسی قرار دهد. این کار از اتلاف زمان زیادی که جهت جابه‌جایی کاراکترها صورت می‌گیرد، جلوگیری می‌کند.
در یکی از این تکنیک‌ها از یک بافر که به دو نیمه N کاراکتری تقسیم شده است، استفاده می‌شود و با یک فرمان read به‌جای خواندن یک کاراکتر می‌توان N کاراکتر ورودی را در هر نیمه بافر وارد کرد.
هر یک از انواع بافرها دارای دو نوع اشاره‌گر می‌باشند:
Forward: با خواندن هر کاراکتر یک واحد به جلو می‌رود.
Lexeme – beginning: ابتدای توکنی را که می‌خواهیم مشخص کنیم، نشان می‌دهد.
در ابتدا هر دو نشانه رو به اولین کاراکتر Lexem بعدی که باید پیدا شود، اشاره می‌کنند.





Forward تا جایی جلو می‌رود که به‌کاراکتری برسد که جزو الگو نیست.

پس از شناسایی توکن، lexeme – beginning به شروع توکن بعدی set می‌شود.

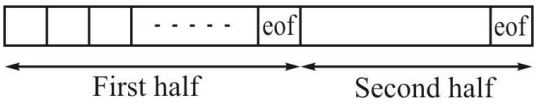
کد مربوط به پیشروی اشاره‌گر پیشرو به شکل زیر خواهد بود:

```
if forward at the end of first half then
  begin
        reload second half;
        forward = forward + 1;
  end
else if forward at the end of second half then
  begin
        reload first half ;
        move forward to beginning of the first half ;
  end
else forward = forward +1;
```

در این روش به‌ازای خواندن هر کاراکتر دو مقایسه نیاز است که این کار زمان‌بر می‌باشد.

برای حل این مشکل، یک نگهبان در آخر هر نیمه بافر قرار می‌دهیم تا تعداد مقایسه‌ها به یکی کاهش پیدا کند.

eof : کاراکتر نگهبان که انتهای هر نیمه بافر قرار می‌گیرد و کاراکتر ویژه‌ای است که جزء متن محسوب نمی‌شود.



کد مربوط به حرکت اشاره‌گر forward همراه با کاراکتر نگهبان به‌صورت زیر می‌باشد:

```
forward = forward + ۱;
if  forward ↑= eof  then
begin
   if forward at the end of first half then
   begin
      reload second half;
      forward=forward + ۱;
   end
   else if forward at the end of second half then
   begin
      reload first half;
      move forward to beginning of first half
   end
   else / *eof whitin a buffer signifiging end of input * /
end
```

الگوی توکن‌ها را توسط عبارات منظم نمایش می‌دهیم و هر عبارت منظمی قابل تبدیل به DFA است.

عبارات منظم (Regular Expressions)

1.$\varnothing$ is a RE

2.λ is a RE

3.$a \in I$ is a RE

4.if r_1 and r_2 are REs then

$r_1 \mid r_2$	$L(r_1) \mid L(r_2)$
$r_1.r_2$	$L(r_1).L(r_2)$
(r_1)	$L(r_1)$
$r_1 *$	$(L(r_1))^*$

are REs, too.

$r.r^* = r^+$

$r.\varepsilon = \varepsilon.r = r$

$(r^*)^* = r^*$

$(a \mid b)^* = (a^* b^*)^* = (\underset{a\mid\varepsilon}{a?}\, b?)^*$

اگر r یک عبارت منظم باشد، دراین‌صورت r? یک عبارت منظم است که زبان $L(r) \cup \{\varepsilon\}$ را توصیف می‌کند.

خاصیت جابه‌جایی: $r \mid s = s \mid r$

خاصیت شرکت‌پذیری: $r \mid (s \mid r) = (r \mid s) \mid r$

خاصیت شرکت‌پذیری اپراتور الحاق: $(r\ s)t = r(s\ t)$

الحاق روی | توزیع‌پذیر است: $(s \mid t)r = sr \mid tr \quad r(s \mid t) = rs \mid rt$

ε عضو بی‌اثر الحاق می‌باشد: $r\varepsilon = r, \varepsilon r = r$

اعداد باینری: $(0 \mid 1)^+$

رشته‌های حداقل به طول 1: $(a \mid b)^*\ (a \mid b)\ (a \mid b)^*$

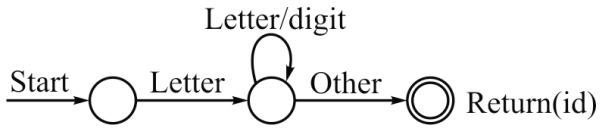
تمام رشته‌هایی که حداکثر دو a متوالی دارند روی $\{a, b\}$: $(aab \mid ab \mid b)^*\ a?a?$

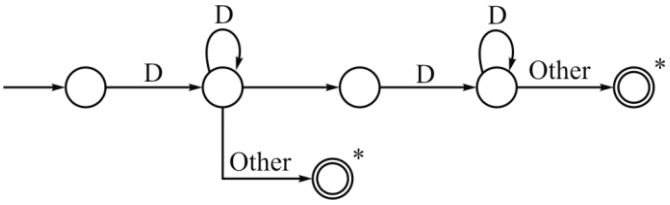
دیاگرام‌های انتقال

یک دیاگرام انتقال یک گراف جهت‌دار است که هر یک از گره‌های آن مشخص‌کننده یک state می‌باشند. برای پیاده‌سازی دستی اسکنر از دیاگرام انتقال کمک می‌گیریم.

با خواندن کاراکترهای یک رشته، تطبیق آن‌ها با برچسب‌های دیاگرام و پیمایش آن دیاگرام می‌توان گفت که آن رشته متعلق به زبان مورد نظر هست یا نه.

مثال: دیاگرام تشخیص شناسه (identifire) به‌صورت زیر می‌باشد:





دیاگرام تشخیص اعداد صحیح و اعداد اعشار

علامت ستاره یعنی اینکه باید آخرین ورودی به بافر برگردد.

الگوی هر توکن را با استفاده از یک ماشین متناهی (DFA یا NFA) نشان می‌دهیم. اگر از دیاگرام انتقال حالت برای تشخیص کلمات کلیدی استفاده شود، تعداد حالات زیاد شده و سرعت پایین می‌آید، پس بهتر است کلمات کلیدی در آغاز کار در جدول سمبل‌ها یا جدول نشانه‌ها قرار داده شوند.

تابع ( ) install به جدول سمبل‌ها نگاه می‌کند، اگر توکن خوانده شده کلمه کلیدی باشد صفر را برمی‌گرداند و اگر کلمه کلیدی نباشد و شناسه باشد، دو حالت داریم: اگر این شناسه در جدول سمبل‌ها وجود داشته باشد اشاره‌گر به مدخلی که این توکن در آن قرار دارد، برگردانده می‌شود و اگر در جدول توکن‌ها نباشد، در جدول سمبل‌ها قرار داده می‌شود و اشاره‌گر به مدخلی که قرار داده می‌شود، برگردانده می‌شود.

تابع (get token) اگر توکن خوانده شده کلمه کلیدی باشد نشانه متناظر آن را برمی‌گرداند و اگر شناسه باشد، id را برمی‌گرداند. برای بالا رفتن سرعت بهتر است توکن‌هایی که بیشتر تکرار می‌شوند، در Caseهای اول و Stateهای آغازین قرار بگیرند.

اسکنر باید همواره طولانی‌ترین توکن ممکن را تشخیص دهد. مثلاً دیاگرام تشخیص اعداد اعشاری باید قبل از دیاگرامی باشد که اعداد صحیح را تشخیص می‌دهد.

اگرچه می‌توان قواعد لغوی زبان‌ها را توسط گرامرهای مستقل از متن نیز بیان کرد، ازآنجاکه این قواعد اغلب ساده‌اند می‌توان آن‌ها را با عبارات منظم بیان کرد و نیازی به نمایش قوی‌تر مستقل از متن نیست و همچنین اسکنرهای سریع‌تری را از روی عبارات منظم می‌توان تولید کرد، همچنین عبارات منظم وسیله‌ای فشرده‌تر و گویاتر از گرامرهای مستقل از متن هستند بنابراین قواعد لغوی زبان‌ها را با عبارات منظم نمایش می‌دهیم.

تشخیص خطا

اسکنر نمی‌تواند خطاهای زیادی را تشخیص دهد زیرا برنامه مبدأ را به‌صورت کاراکتر به‌کاراکتر می‌خواند و دیدگاه محلی نسبت به برنامه دارد. اگر رشته‌ای از کاراکترها با الگوی هیچ‌کدام از توکن‌ها مطابقت نداشته باشد، در این‌صورت خطای لغوی رخ می‌دهد و اسکنر باید این قابلیت را داشته باشد که از این خطا و خطاهای دیگر گذر کرده و عمل Scan (تحلیل لغوی) را تا پایان فایل ادامه دهد.

در برخورد با خطاهای لغوی به دو روش عمل می‌شود:

۱- بازیافت خطا (پوشش خطا، Error recovery)

۲- تصحیح خطا (Error Correction)

بازیافت خطا

روش Panic Mode: اگر رشته ورودی ما a b c $ e f باشد، هرگاه forward به‌کاراکتر غیر مجاز $ رسید تمام توکن‌ها از ابتدا تا کاراکتر غیر مجاز حذف می‌شود و بقیه را توکن در نظر می‌گیرد یعنی lexeme – begin روی "e" قرار می‌گیرد. اشکال این روش بروز خطاهای آبشاری (Cascaded Errors) است.

- تصحیح خطا
- جابه‌جایی دو کاراکتر $=:\rightarrow:=$
- حذف کاراکتر غلط $abc\$ef \rightarrow abcef$
- اضافه کردن یک کاراکتر جدید (کاراکتر گمشده) $:\rightarrow:=$
- تغییر یک کاراکتر
- تعویض یا جایگزینی دو کاراکتر مجاور مثل $><\rightarrow<>$

تحلیل نحوی (Syntax Analysis)

در مرحله تحلیل نحوی، برنامه ورودی از لحاظ دستوری چک می‌شود.

وظیفه اصلی parser این است که تشخیص دهد که رشته توکن‌های تولید شده توسط اسکنر، آیا توسط قواعد زبان قابل تولید هست یا نه.

مروری بر نظریه زبان‌ها و ماشین‌ها

ماشین خودکار متناهی: ابزاری برای تشخیص ساختارهای موجود زبان در دنباله ورودی از نشانه‌ها و پذیرفتن یا نپذیرفتن دنباله کاراکترهای ورودی می‌باشد. انواع ماشین‌های خودکار عبارتند از:

۱- ماشین خودکار متناهی قطعی (DFA)

۲- ماشین خودکار متناهی غیرقطعی (NFA)

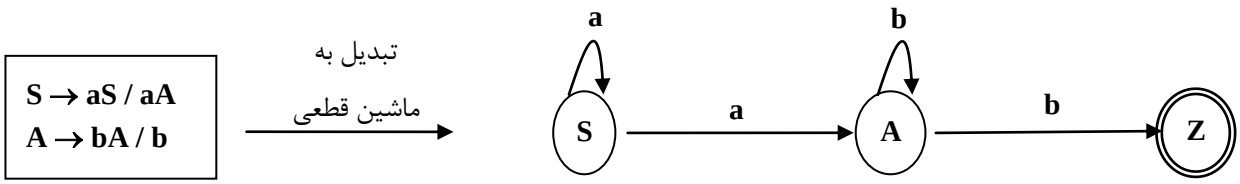
ماشین خودکار قطعی یک مدل ریاضی است متشکل از 5 تایی به‌صورت زیر می‌باشد:

$$m = (Q, \Sigma, \delta, q_\circ, F)$$

که در آن Q مجموعه متناهی از حالات ماشین، Σ الفبای زبان، δ تابع انتقال، $q_\circ$ حالت شروع ماشین و F زیرمجموعه‌ای از Q به‌نام حالات نهایی می‌باشد.

ماشین خودکار غیر قطعی: ماشین متناهی است که می‌توان از هر حالت با عناصر ورودی یکسان به حالات مختلفی رسید.

مثال: نمونه‌ای از DFA به‌صورت زیر می‌باشد:



رشته پردازش شده	محاسبه	اشتقاق
A	$[S, aabb] \rightarrow [S, abb]$	$S \rightarrow aS$
Aa	$[A, bb]$	$\rightarrow aaA$
Aab	$[A, b]$	$\rightarrow aabA$
Aabb	$[Z,]$	$\rightarrow aabb$

تبدیل NFA به DFA

همبستگی لاندا یا $\lambda\text{-closure}(q_i)$ به‌صورت بازگشتی زیر تعریف می‌شود:

۱- پایه: $q_i \in \lambda\text{-closure}(q_i)$

۲- مرحله بازگشت: اگر q_i یک عنصر از $\lambda\text{-closure}(q_i)$ باشد و اگر $q_k \in \delta(q_i, \lambda)$ آنگاه: $q_k \in \lambda\text{-closure}(q_i)$

۳- همبستگی: q_j در $\lambda\text{-closure}(q_i)$ است اگر بتواند با تکرار متناهی از مرحله بازگشت فوق به‌دست آید.

نکات تستی فصل اول

۱- اگر کاراکتر غیر مجاز در متن برنامه نوشته شود، **تحلیلگر لغوی** آن تشخیص داده است.

۲- در زبان پاسکال عبارت If(a+b then به‌دلیل عدم توازن پرانتزها دارای **خطای نحوی** است.

۳- در زبان پاسکال عبارت A B:= به‌دلیل عدم رعایت ترتیب صحیح انتساب دارای **خطای نحوی** است.

۴- کشف خطا مربوط به ساختار تک تک لغات وظیفه **تحلیلگر لغوی** است.

۵- درصورتی‌که شناسه قبلا اعلان نشده باشدِ یک **خطای معنایی** رخ می‌دهد.

۶- در مرحله **تحلیل لغوی** کلیه شناسه‌های موجود در برنامه وارد جدول علائم (Symob Table) می‌شود.

۷- تحلیلگر معنایی، معنی‌دار بودن عباراتی که از نظر نحوی درست بوده‌اند را مورد بررسی قرار می‌دهد.

سوالات چهارگزینه‌ای سراسری فصل اول

۱- با در نظر گرفتن گرامر مقابل، کدام‌یک از گزینه‌های زیر صحیح است؟ **(سال ۸۲)**

$$E \to T + E \mid T * E \mid T$$
$$T \to P - T \mid P$$
$$P \to F / P \mid E$$
$$F \to id \mid (E)$$

۱) عملگرها دارای شرکت‌پذیری راست (Right associative) هستند و عملگر منها اولویت بیشتری نسبت به عملگر تقسیم دارد.

۲) عملگرها دارای شرکت‌پذیری راست (Rigth associative) هستند و عملگر منها اولویت کمتری نسبت به عملگر تقسیم دارد.

۳) عملگرهای ضرب و جمع اولویت بیشتری از عملگرهای دیگر دارند و عملگرها از چپ شرکت‌پذیر (Left associative) هستند.

۴) عملگرهای ضرب و جمع تقدم کمتری نسبت به سایر عملگرها دارند و عملگرها از سمت چپ شرکت‌پذیر (Left associative) هستند.

۲- در برنامه زیر هر دو کاربرد عدد ۱۰/۵ خطا است. خطای اول در Declaration و خطای دوم در عبارت به ترتیب در کدام‌یک از بخش‌های Compiler کشف می‌شوند؟ **(سال ۸۳)**

```
  ⋮
int A[10.5];
  ⋮
...A[10.5] + B...
  ⋮
```

۱) Parser و Parser

۲) Parser و Scanner

۳) Parser و کد خطایاب در زمان اجرا

۴) Parser و یکی از Semantic routineها

۳- می‌دانیم که هر زیر برنامه می‌تواند دارای تعدادی پارامتر باشد و هنگام فراخوانی زیر برنامه تعداد آرگومان‌ها باید با تعداد پارامترها مطابقت داشته باشد. اگر در برنامه‌ای این مطابقت رعایت نشده باشد خطای مربوطه در کدام‌یک از مراحل زیر ردیابی می‌شود؟ **(سال ۸۴)**

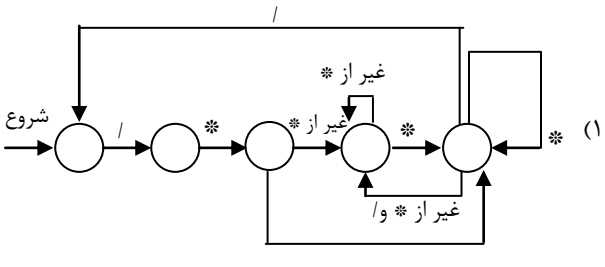
۱) زمان اجرا (Run Time)

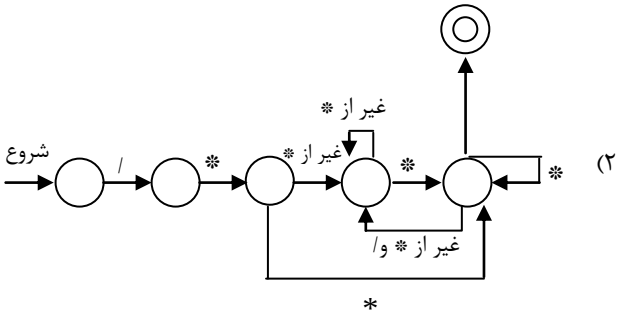
۲) تحلیل نحوی (Syntax Analysis)

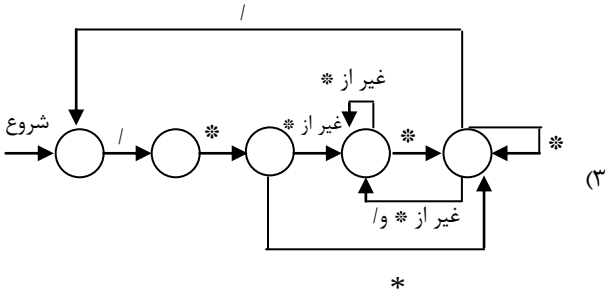
۳) تحلیل معنایی (Semantic Analysis)

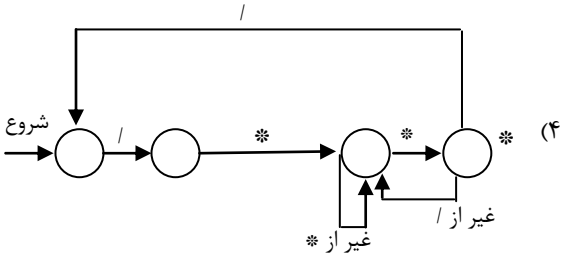
۴) تولید کد (Code Generation)

۴- در یک زبان برنامه‌سازی Commentها با /* شروع و با */ تمام می‌شوند. کدام‌یک از موارد زیر عملکرد Scanner را در نادیده گرفتن Commentها بهتر نمایش می‌دهد؟ (سال ۸۸)

۱)


۲)


۳)


۴)


۵- کامپایلرهای A و B برای زبان X مفروضند. کامپایلر A به ازای ارجاع به آرایه‌های یک بعدی دو کد به زبان ماشین و کامپایلر B سه کد تولید می‌کند. کامپایلر A برای اصلاح توصیف‌کننده هر آرایه، محاسباتی انجام می‌دهد ولی B کار خاصی نمی‌کند. کدام گزینه صحیح است؟ (سال ۸۸)

۱) فضایی که کامپایلر A برای نگهداری اطلاعات آرایه‌ها در زمان اجرا اختصاص می‌دهد از B کمتر است.

۲) کامپایلر A در مجموع برای آماده‌سازی توصیف‌کننده آرایه و ترجمه هر ارجاع به آرایه یک بعدی وقت بیشتری می‌گیرد ولی B کدی تولید می‌کند که اجرای آن سریع‌تر است.

۳) هر دو کامپایلر برای یک برنامه واحد با هر تعداد ارجاع به آرایه‌های یک بعدی به یک اندازه وقت می‌گیرند ولی کدی که A تولید می‌کند سریع‌تر اجرا می‌شود.

۴) کامپایلر B در مجموع برای آماده‌سازی توصیف‌کننده آرایه و ترجمه هر ارجاع به آرایه یک بعدی وقت بیشتری می‌گیرد ولی A کدی تولید می‌کند که اجرای آن سریع‌تر است.

۶- خطاهای زیر در برنامه‌های یک زبان برنامه‌سازی متعارف مفروض است. کدام دسته از خطاها توسط پارسر قابل کشف هستند؟
(سال ۹۲)

I- نابرابری تعداد اندیس‌های یک آرایه با تعداد ابعاد تعریف شده آرایه
II- ناهمخوانی نوع متغیرهای A و B در عبارت A + B
III- نابرابری تعداد پارامترهای فراخوانی یک تابع، با تعداد پارامترهای تعریف شده برای آن
IV- ناهمخوانی نوع یک پارامتر در فراخوانی یک تابع، با نوع تعریف شده آن در تعریف تابع

۱) I و III ۲) II و IV ۳) I، II، III و IV ۴) هیچ‌کدام

۷- استفاده از = به‌جای = = در شرط‌های زبان C، چه نوع خطایی محسوب می‌شود؟ **(سال ۹۲)**

۱) ساختاری (Syntactic) ۲) معنایی (Semantic)
۳) منطقی (Logical) ۴) لغوی (Lenical)

پاسخنامه سوالات چهارگزینه‌ای سراسری فصل اول

۱- گزینه «۲»

☜گرامر بازگشتی راست می‌باشد سپس عملگرها دارای شرکت‌پذیری راست هستند. از طرفی قاعده $P \Rightarrow F / P$ تقدم بر قاعده $T \Rightarrow P - T$ گسترش می‌یابد در نتیجه اولویت عملگر تقسیم بیشتر از اولویت عملگر منها است.

۲- گزینه «۴»

☜خطای Declaration توسط پارسر گرفته می‌شود و خطای دوم که اندیس آرایه به‌صورت اعشار بیان شده است، توسط یکی از Semantic routinها گرفته می‌شود. در ضمن اگر خطای اندیس آرایه خارج از محدوده رخ دهد در آن صورت خطای به‌وجود آمده، توسط برنامه‌ای که حاوی دستور است، گرفته می‌شود.

۳- گزینه «۳»

☜تطابق تعداد پارامترها و همچنین نوع آن‌ها به‌هنگام تحلیل معنایی مشخص می‌شود. توجه کنید که فراخوانی زیر برنامه دارای یک گرامر متفاوت با خود زیر برنامه است و این مسئله با تحلیل نحوی قابل تشخیص نیست.

۴- گزینه «۱»

☜بعد از عبور از یک Comment، کامپایلر باید به حالتی که قبل از آن بوده است، برگردد بنابراین گزینه ۲ نمی‌تواند صحیح باشد. ماشین گزینه ۴ با دیدن ساختار /*** به حالت اولیه خود برنمی‌گردد و بنابراین این رشته را به‌عنوان یک Comment نمی‌شناسد بنابراین این گزینه نیز نادرست می‌باشد. همچنین ماشین گزینه ۳ رشته /*BA* را به‌عنوان یک Comment نمی‌شناسد، بنابراین این گزینه نیز نمی‌تواند صحیح باشد.

۵- گزینه «۴»

☜زیرا کامپایلر A برای اصلاح توصیف‌کننده هر آرایه، محاسباتی را انجام می‌دهد ولی کامپایلر B برای آماده‌سازی توصیف می‌کند. آرایه به زمانی در زمان اجرا نیاز دارد.

۶- گزینه «۴»

۷- گزینه «۳»